Profiling Projects 25

Profiling your program lets you analyze how it runs. A profiler can log how much time your program spends in each function, and it can trace your program by printing the name of each routine as it's called. This chapter shows you how to use the profiler library that comes with the THINK Project Manager.

Contents

Using the Profiler 507
    Setting options to profile the most functions possible . . 508

Logging Time Statistics 509

Tracing Your Functions 510

Modifying the Profiler 511
    Changing where to print 511
    Changing which timer to use 512
    Changing the profiler's report 512
    Recompiling the profiler 513

Summary 514
    User functions 514
    Internal functions 515

25 Profiling Projects

Using the Profiler

The profiler helps you analyze how your program runs. It can log how much time your program spends in each function and trace your program by printing the name of each routine as it's called. It works with both THINK C and Symantec C++.

To use the profiler, follow these steps:

1. Check the Generate profiler calls check box in the Debugging page of the options dialog for the compiler your project uses: either THINK C or Symantec C++. When this option is on, the compiler inserts calls to profiler routines at the beginning and end of your functions.

2. If you're using THINK C, include the Profile library and the ANSI library in your project. If you're using Symantec C++, include the Profile++ library and the ANSI++ library in your project. They're in the `Standard Libraries` folder. Make sure you include `profile.h` in the file that calls `InitProfile()`.

3. Call `InitProfile()` in your program to start the profiler. This function takes two arguments. The first `nsyms` is the maximum number of functions that the profiler will track. The second `depth` is the maximum call depth. A good default for both arguments is 200 (for example, `InitProfile(200,200)`).

4. Set the variables `_profile` and `_trace` to control what the profiler does. `InitProfile()` initially sets `_profile` to non-zero and `_trace` to zero. If `_profile` is non-zero, the profiler logs time statistics. If `_trace` is non-zero, the profiler traces your routines.

5. If you want to see the timing statistics, make sure the profiler prints them out. You must call `DumpProfile()` or `exit()`, or use the console environment.

For example, this program initializes the profiler, writes a profile, and exits:

```
#include <profile.h>

void main()
{
   /* Initialize the profiler with the */
   /* recommended arguments.           */
   InitProfile( 200, 200 );

   /* Have the profiler write a trace  */
   /* of your code.                    */
   _trace = 1;

   /* Do some stuff.                   */
   . . .

   /* Write a profile.                 */
   DumpProfile();

   /* Do some more stuff.              */
   . . .

   /* Exit. The profiler writes a      */
   /* profile automatically.           */
   exit(0);
}
```

Profiling a C++ program can greatly increase the amount of global data it contains. You may need to resegment your project so there will be enough room in each segment. You can turn on and off profiling for a particular function or group of functions with the `#pragma trace on` and `#pragma trace off` statements, described in the *Symantec C++ Compiler Guide*. You can place the profiling information in the program's code instead of in its global data space by turning on the Place String Literals in Code option in the Compiler Settings page of the Symantec C++ options dialog. However, the profiling information will be as large as before.

Setting options to profile the most functions possible

If you're using THINK C, the profiler won't collect statistics for some unusual functions. These conditions control whether the profiler can profile a function:

- Local Variables. Does the function declare any local variables?

- Function Parameters. Does the function declare any parameters?
- Inline Assembly. Does the function use inline assembly?
- Stack Frames option. Is the Always Generate Stack Frames option in the Debugging page of the THINK C options dialog on?

When the conditions for a function match one of the cases in the table below, the profiler won't collect statistics for the function:

Local Variables	Function Parameters	Inline Assembly	Stack Frames Option
none	none	present	on or off
none	none	none	off

Table 25-1 The functions for which the profiler won't collect statistics

To find out why the profiler doesn't profile these functions, see "Debugging Programs" in Chapter 12.

For Symantec C++, the profiler also does not profile inlined C++ functions, unless you choose the Use Function Calls for Inlines option in the Debugging page of the Symantec C++ options dialog. (THINK C does not support inlined functions.)

Logging Time Statistics

The profiler can log the amount of time spent in each function. To print these results to `stdout` at any time, call `DumpProfile()`. The profiler automatically prints out these results, if your program does one of these when it exits:

For more information on the console environment, see the online Standard Libraries Reference.

- Calls `exit()` to exit
- Uses the console environment

Note

If your program exits by calling `ExitToShell()` or `abort()`, the profiler doesn't print its statistics automatically.

For more information on `exit()` *and* `abort()`*, see the online Standard Libraries Reference.*

The profiler uses the units of the VIA timer. Each unit is 1.2766 µsec (approximately 780,000 in a second). You can change the units to ticks (60ths of a second) by editing the profiler code. You might

need to use the tick counter if your program uses the Sound Driver, since the Sound Driver uses the VIA timer. Also, the VIA timer is less accurate if there's a lot of interrupt-level activity. See "Changing which timer to use" later in this chapter.

This is an example of the profiler report. It's for a program that compares two sorting algorithms.

```
Function      Minimum  Maximum  Average    %   Entries

compare           26      238       27     4    12948
std_swap          64      948       67     2     2353
SlowSort         582  4880405  1643130    68        3
QuickSort         61       61       61     0        3
_QuickSort        57       58       57     0        3
QuickSort1        27   177573     1051     8      564
std_compare       85      436       88    15    12948
TryAlgorithms    364    24996     9294     0        3
```

The following table describes the report's headings:

Heading	Description
Function	Name of the function.
Minimum	Minimum time spent in routine.
Maximum	Maximum time spent in routine.
Average	Average time spent in routine.
%	Percentage of profiling period spent in the routine. (The profiling period is the accumulated time spent in routines that were compiled with the Generate Profiler Calls Option on.)
Entries	Number of times the routine was called.

Table 25-2 Profiler report headings

If you don't want the profiler to keep these statistics, set the variable `_profile` to `0` after you call `InitProfile()`.

Tracing Your Functions

The profiler can also print a trace of the functions your program calls as it runs. Every time you call a function, the profiler prints out the

function's name, indenting it to show how far nested it is. For example, this is an excerpt from the trace of the sorting program:

```
TryAlgorithms
 SlowSort
 QuickSort
  _QuickSort
   QuickSort1
    std_compare
     compare
    std_compare
     compare
    . . .
```

According to this trace, `TryAlgorithms()` called `SlowSort()` and `QuickSort()`, then `QuickSort()` called `_QuickSort()`, and so on.

If you want the profiler to trace your functions, set the variable `_trace` to a non-zero value after you call `InitProfile()`.

Modifying the Profiler

This section describes how to change where the profiler prints its reports and which timer the profiler uses.

Changing where to print

For more information on `cecho2printer()`, `cecho2file()`, *and* `freopen()`, *see the online Standard Libraries Reference.*

The profiler prints its report and trace to `stdout`. By default, `stdout` is a console window, a simple window that acts like a glass terminal. You can have the profiler print to both a console and a file, to both a console and a printer, or to just a file.

To print to…	Use this function…
a console and a file	`cecho2file()`
a console and a printer	`cecho2printer()`
just a file	`freopen()`

Table 25-3 Choosing where the profiler prints

For example, these statements initialize the profiler and redirect `stdout` to the file `profiler report`:

```
InitProfile( 200, 200 );
freopen( "profiler report", "w", stdout );
```

Changing which timer to use

You can choose whether the profiler uses the VIA timer or the tick counter. The profile libraries shipped with the THINK Project Manager uses the VIA timer.

Usually you'll want to use the more precise VIA timer. Each of its units is 1.2766 µsec (approximately 780,000 in a second) and each tick unit is one sixtieth of a second. However, you may need to use the tick counter in these two cases:

- When you use the Sound Driver. The Sound Driver uses the VIA timer and will interfere with the profiler.
- When your program has a lot of interrupt-level activity. The VIA timer is less accurate in this case.

To create a profile library that uses the tick counter, follow the steps in the "Recompiling the profiler" later in this chapter. When you come to the step "Make your changes," comment out the line:

```
#define _VIATIMER_
```

Changing the profiler's report

You can change the profiler's report. For example, you might alter how a field is computed or what it looks like. Follow the steps in the "Recompiling the profiler" later in this chapter. When you come to the step "Make your changes," edit the function `DumpProfile()`.

You'll need to edit two statements in `DumpProfile()`. The first is the `printf()` statement that prints a line of information for each function profiled. Below is an example with comments explaining the arguments:

```
printf("%#-32s%8lu %8lu %8lu   %3u %8lu\n",
   p->fname,                                  // Name
   p->min == LONG_MAX ? 0 : p->min,           // Min
   p->max,                                    // Max
   p->count ? p->total / p->count : 0,        // Ave
   total ? (int) (100. * p->total/total) : 0,
                                              // %
   p->count);                                 // Ent.
```

The second is the `printf()` statement that prints the report's heading. You need to edit it only if you change the length of a field, as shown below:

```
printf("\n%-32s  Minimum  Maximum  "
       "Average    %%   Entries\n\n",
       "Function");
```

This example shows how to print the percentage as a floating point number with five digits of accuracy, instead of as an integer. First, edit the format string argument in the first `printf()` statement. Since the percentage will be a floating point number, change its format specifier from `%3u` to `%5.2f`, as in the following example:

```
printf("%#-32s%8lu %8lu %8lu   %5.2f %8lu\n",
```

Second, change the argument that computes the percentage. Simply delete the `(int)` cast, which casts the argument to an integer, and change `0` to `0.0`, as shown below:

```
   total ? (100.* p->total/total) : 0.0,
                                    /* % */
```

Finally, add some spaces to the report's heading, since the percentage field is longer. Add three spaces between `%%` and `Entries`, as shown below:

```
printf("\n%-32s  Minimum  Maximum  "
       "Average    %%      Entries\n\n",
       "Function");
```

Recompiling the profiler

To change how the profiler works, you need to edit and recompile the profile project. Follow these steps:

1. In the Finder, create a new copy of the project. If you plan to use the new profiler with THINK C projects, copy the Profile project. If you plan to use the new profiler with Symantec C++ projects, copy the Profile++ library.
2. Give the project a name, like `new profile`.
3. Open the new project with the THINK Project Manager.
4. Open the file `profile.c`.
5. Make your changes.

6. Choose the **Save As** command. Go to the `C Sources` folder in the `Standard Libraries` folder. Save the file under a new name, such as `new profile.c`.

7. Choose the **Compile** command to recompile the file.

Warning

When you rebuild the profiler library, make sure that the Generate Profiler Calls option is not selected. You cannot profile the profiler.

Now you can use the new profiler as a library in any of your projects.

Summary

This section describes the functions and variables in the Profile and Profile++ libraries. They're declared in `profile.h`.

User functions

Use these functions and variables in your program to control how the profiler works.

InitProfile

```
void InitProfile( unsigned nsyms,
                  unsigned depth );
```

Start profiling. It takes these arguments:

Argument	Description
`nsyms`	The maximum number of functions to track
`depth`	The maximum call depth

DumpProfile

```
void DumpProfile(void);
```

Writes the statistics collected so far to `stdout`. You can call this function any time to print the current statistics. The profiler automatically calls it when your program exits.

_profile

```
short int _profile;
```

Variable. If non-zero, the profiler collects timing statistics as your functions execute. By default, `_profile` is non-zero. You can change the value after you call `InitProfile()`.

_trace

```
short int _trace;
```

Variable. If non-zero, the profiler traces your functions. Every time you enter a function, the profiler prints its name to `stdout`. By default, `_trace` is zero. You can change the value after you call `InitProfile()`.

Internal functions

The profiler calls these functions before and after each function call. You shouldn't call them in your programs.

profile

```
void _profile_( unsigned char *fname );
```

Called at the beginning of each function when the Generate Profiler Calls option is on. When this routine is called, the stack contains the address of a Pascal string that is the name of the function. `_profile_()` changes the return address of the routine to be `_profile_exit()`.

_profile_exit

```
static void *_profile_exit(void);
```

Called at the end of a function. Code to execute this routine is not generated. Instead, `_profile_()` changes the stack so the original routine "returns" to this one.